Codebase Refactoring Tool with Neo4j Graph Database

Overview

A desktop application that creates an Abstract Syntax Tree (AST) representation of large codebases, stores it in Neo4j, and enables intelligent refactoring without overwhelming LLM context windows.

Architecture

Core Components

1. **File System Scanner**
   - Recursively traverse project directories
   - Build initial file tree structure
   - Filter by file extensions and ignore patterns
2. **AST Parser Engine**
   - Multi-language support (Python, JavaScript, Java, etc.)
   - Extract classes, methods, functions, and their relationships
   - Identify function calls and dependencies
3. **Neo4j Graph Builder**
   - Convert AST to graph nodes and relationships
   - Store file hierarchy, code structure, and dependencies
   - Maintain bidirectional relationships for traversal
4. **Linting Integration**
   - Run language-specific linters
   - Attach lint errors/warnings to specific nodes
   - Enable filtering by error severity
5. **Graph Visualization & Pruning**
   - Interactive tree visualization
   - Node selection and subtree pruning
   - Context window management
6. **Refactoring Engine**
   - Graph traversal algorithms
   - Pattern matching for code issues
   - Automated fix generation and propagation

Implementation Details

Technology Stack

- **Backend**: Python (for AST parsing and graph operations)
- **Database**: Neo4j
- **Frontend**: Electron + React (for desktop UI)
- **Visualization**: D3.js or Cytoscape.js
- **AST Parsers**:
  - Python: `ast` module
  - JavaScript: `@babel/parser`
  - Java: `JavaParser`
  - Multi-language: `tree-sitter`

Graph Schema

```cypher
```

```
// Node Types
(:File {path, name, extension, size})
(:Class {name, line_start, line_end})
(:Method {name, parameters, return_type, line_start, line_end})
(:Function {name, parameters, return_type, line_start, line_end})
(:LintError {type, message, severity, line})

// Relationships
(:File)-[:CONTAINS]->(:Class)
(:Class)-[:HAS_METHOD]->(:Method)
(:File)-[:CONTAINS]->(:Function)
(:Method)-[:CALLS]->(:Method|Function)
(:Function)-[:CALLS]->(:Method|Function)
(:Method|Function)-[:HAS_ERROR]->(:LintError)
```

Core Python Implementation

```
python
```

```python
import os
import ast
from py2neo import Graph, Node, Relationship
import networkx as nx
from typing import Dict, List, Set
from dataclasses import dataclass
from pathlib import Path

@dataclass
class CodeEntity:
    name: str
    type: str
    file_path: str
    line_start: int
    line_end: int
    calls: Set[str] = None

class CodebaseAnalyzer:
    def __init__(self, neo4j_uri: str, username: str, password: str):
        self.graph = Graph(neo4j_uri, auth=(username, password))
        self.entities: Dict[str, CodeEntity] = {}

    def scan_codebase(self, root_path: str, extensions: List[str] = ['.py', '.js', '.java']):
        """Scan codebase and build file tree"""
        for root, dirs, files in os.walk(root_path):
            # Skip hidden directories and common ignore patterns
            dirs[:] = [d for d in dirs if not d.startswith('.') and d not in ['node_modules', '__pycache__']]

            for file in files:
                if any(file.endswith(ext) for ext in extensions):
                    file_path = os.path.join(root, file)
                    self._process_file(file_path)

    def _process_file(self, file_path: str):
        """Process individual file and extract AST"""
        file_node = Node("File", path=file_path, name=os.path.basename(file_path))
        self.graph.create(file_node)

        if file_path.endswith('.py'):
            self._parse_python_file(file_path, file_node)
        # Add other language parsers here

    def _parse_python_file(self, file_path: str, file_node: Node):
        """Parse Python file and extract entities"""
        with open(file_path, 'r', encoding='utf-8') as f:
            try:
                tree = ast.parse(f.read())
                visitor = PythonASTVisitor(file_path, self.graph, file_node)
                visitor.visit(tree)
                self.entities.update(visitor.entities)
            except SyntaxError as e:
                # Create lint error node
                error_node = Node("LintError",
                            type="SyntaxError",
                            message=str(e),
                            severity="error",
                            line=e.lineno)
                self.graph.create(error_node)
                self.graph.create(Relationship(file_node, "HAS_ERROR", error_node))

class PythonASTVisitor(ast.NodeVisitor):
    def __init__(self, file_path: str, graph: Graph, file_node: Node):
        self.file_path = file_path
        self.graph = graph
        self.file_node = file_node
        self.entities: Dict[str, CodeEntity] = {}
```

```
        self.current_class = None

    def visit_ClassDef(self, node: ast.ClassDef):
        class_node = Node("Class",
                          name=node.name,
                          line_start=node.lineno,
                          line_end=node.end_lineno)
        self.graph.create(class_node)
        self.graph.create(Relationship(self.file_node, "CONTAINS", class_node))

        old_class = self.current_class
        self.current_class = class_node
        self.generic_visit(node)
        self.current_class = old_class

    def visit_FunctionDef(self, node: ast.FunctionDef):
        func_type = "Method" if self.current_class else "Function"
        func_node = Node(func_type,
                         name=node.name,
                         parameters=[arg.arg for arg in node.args.args],
                         line_start=node.lineno,
                         line_end=node.end_lineno)
        self.graph.create(func_node)

        if self.current_class:
            self.graph.create(Relationship(self.current_class, "HAS_METHOD", func_node))
        else:
            self.graph.create(Relationship(self.file_node, "CONTAINS", func_node))

        # Extract function calls
        call_visitor = CallVisitor()
        call_visitor.visit(node)

        entity = CodeEntity(
            name=node.name,
            type=func_type,
            file_path=self.file_path,
            line_start=node.lineno,
            line_end=node.end_lineno,
            calls=call_visitor.calls
        )
        self.entities[f"{self.file_path}:{node.name}"] = entity

        self.generic_visit(node)

class CallVisitor(ast.NodeVisitor):
    def __init__(self):
        self.calls = set()

    def visit_Call(self, node: ast.Call):
        if isinstance(node.func, ast.Name):
            self.calls.add(node.func.id)
        elif isinstance(node.func, ast.Attribute):
            self.calls.add(node.func.attr)
        self.generic_visit(node)

class RefactoringEngine:
    def __init__(self, graph: Graph):
        self.graph = graph

    def prune_tree(self, root_node_id: str, max_depth: int = 3) -> nx.DiGraph:
        """Prune the graph to a manageable subtree"""
        query = f"""
        MATCH path = (n)-[*0..{max_depth}]->(m)
        WHERE id(n) = {root_node_id}
        RETURN path
        """
```

```
        result = self.graph.run(query)

        # Build NetworkX graph from Neo4j results
        nx_graph = nx.DiGraph()
        for record in result:
            path = record['path']
            for i in range(len(path.nodes) - 1):
                nx_graph.add_edge(path.nodes[i], path.nodes[i+1])

        return nx_graph

    def find_refactoring_candidates(self, pattern: str) -> List[Dict]:
        """Find nodes matching refactoring patterns"""
        # Example: Find long methods
        query = """
        MATCH (m:Method)
        WHERE m.line_end - m.line_start > 50
        RETURN m, m.line_end - m.line_start as length
        ORDER BY length DESC
        """
        return list(self.graph.run(query))

    def apply_refactoring(self, node_id: str, refactor_type: str):
        """Apply specific refactoring to a node and propagate changes"""
        # This would integrate with actual code modification tools
        # and use graph traversal to find all affected locations
        pass

# Desktop Application Bridge
class CodebaseRefactorAPI:
    """API for Electron frontend"""
    def __init__(self, neo4j_config):
        self.analyzer = CodebaseAnalyzer(**neo4j_config)
        self.refactoring = RefactoringEngine(self.analyzer.graph)

    def scan_project(self, path: str):
        self.analyzer.scan_codebase(path)
        return {"status": "success", "entities": len(self.analyzer.entities)}

    def get_graph_data(self):
        """Return graph data for visualization"""
        query = """
        MATCH (n)
        OPTIONAL MATCH (n)-[r]->(m)
        RETURN n, r, m
        """
        results = self.analyzer.graph.run(query)

        nodes = []
        edges = []
        seen_nodes = set()

        for record in results:
            if record['n'] and record['n'].identity not in seen_nodes:
                nodes.append({
                    'id': record['n'].identity,
                    'label': record['n'].get('name', 'Unknown'),
                    'type': list(record['n'].labels)[0]
                })
                seen_nodes.add(record['n'].identity)

            if record['r'] and record['m']:
                edges.append({
                    'source': record['n'].identity,
                    'target': record['m'].identity,
                    'type': type(record['r']).__name__
                })
```

```
    return {'nodes': nodes, 'edges': edges}
```

Electron Main Process

```javascript
const { app, BrowserWindow, ipcMain } = require('electron');
const { spawn } = require('child_process');
const path = require('path');

let mainWindow;
let pythonProcess;

function createWindow() {
  mainWindow = new BrowserWindow({
    width: 1400,
    height: 900,
    webPreferences: {
      nodeIntegration: false,
      contextIsolation: true,
      preload: path.join(__dirname, 'preload.js')
    }
  });

  mainWindow.loadFile('index.html');
}

app.whenReady().then(() => {
  createWindow();

  // Start Python backend
  pythonProcess = spawn('python', ['backend/main.py']);

  pythonProcess.stdout.on('data', (data) => {
    console.log(`Python: ${data}`);
  });
});

// IPC handlers for frontend communication
ipcMain.handle('scan-project', async (event, projectPath) => {
  // Call Python API
  const response = await fetch('http://localhost:5000/api/scan', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ path: projectPath })
  });
  return response.json();
});

ipcMain.handle('get-graph', async () => {
  const response = await fetch('http://localhost:5000/api/graph');
  return response.json();
});
```

React Frontend Component

```jsx
```

```
import React, { useState, useEffect } from 'react';
import { ForceGraph2D } from 'react-force-graph';
import { Panel, Button, Tree } from '@blueprintjs/core';

function CodebaseVisualizer() {
 const [graphData, setGraphData] = useState({ nodes: [], links: [] });
 const [selectedNode, setSelectedNode] = useState(null);
 const [pruneDepth, setPruneDepth] = useState(3);

 useEffect(() => {
  loadGraph();
 }, []);

 const loadGraph = async () => {
  const data = await window.api.getGraph();
  setGraphData(data);
 };

 const handleNodeClick = (node) => {
  setSelectedNode(node);
 };

 const pruneTree = async () => {
  if (!selectedNode) return;

  const prunedData = await window.api.pruneTree(selectedNode.id, pruneDepth);
  setGraphData(prunedData);
 };

 const runRefactoring = async () => {
  if (!selectedNode) return;

  const result = await window.api.refactor(selectedNode.id);
  // Handle refactoring results
 };

 return (
  <div className="app-container">
   <div className="sidebar">
    <Panel>
     <h3>Controls</h3>
     <Button onClick={() => window.api.scanProject()} text="Scan Project" />
     <div className="prune-controls">
      <label>Prune Depth: {pruneDepth}</label>
      <input
       type="range"
       min="1"
       max="10"
       value={pruneDepth}
       onChange={(e) => setPruneDepth(e.target.value)}
      />
      <Button onClick={pruneTree} text="Prune Tree" />
     </div>
     {selectedNode && (
      <div className="node-details">
       <h4>Selected: {selectedNode.label}</h4>
       <p>Type: {selectedNode.type}</p>
       <Button onClick={runRefactoring} text="Refactor" intent="primary" />
      </div>
     )}
    </Panel>
   </div>

   <div className="graph-container">
    <ForceGraph2D
     graphData={graphData}
```

```
      onNodeClick={handleNodeClick}
      nodeLabel="label"
      nodeColor={node => {
       const colors = {
        'File': '#3182ce',
        'Class': '#38a169',
        'Method': '#d69e2e',
        'Function': '#e53e3e',
        'LintError': '#e53e3e'
       };
       return colors[node.type] || '#718096';
      }}
     />
    </div>
   </div>
  );
}
```

Key Features

1. Intelligent Context Management

- Graph-based representation allows selective loading of code context
- Prune tree to specific areas of interest
- Maintain relationships while keeping context window manageable

2. Multi-Language Support

- Pluggable parser architecture
- Language-specific AST handling
- Unified graph representation

3. Refactoring Algorithms

- Pattern-based detection (long methods, code duplication, etc.)
- Graph traversal for impact analysis
- Automated fix propagation across codebase

4. Visual Navigation

- Interactive graph visualization
- Zoom and pan capabilities
- Filter by node type, lint errors, or custom criteria

5. LLM Integration (Optional)

- Use pruned subgraphs as context for LLMs
- Generate refactoring suggestions
- Validate proposed changes

Installation & Setup

Prerequisites

- Python 3.8+
- Node.js 14+
- Neo4j 4.0+

Installation Steps

```bash
```

```bash
# Clone repository
git clone https://github.com/your-repo/codebase-refactor-tool
cd codebase-refactor-tool

# Install Python dependencies
pip install -r requirements.txt

# Install Node dependencies
npm install

# Start Neo4j database
neo4j start

# Configure Neo4j connection in config.json
{
  "neo4j": {
    "uri": "bolt://localhost:7687",
    "username": "neo4j",
    "password": "your-password"
  }
}

# Run the application
npm start
```

Usage Workflow

1. **Open Project**: Select root directory of codebase
2. **Scan & Parse**: Tool builds AST and populates Neo4j
3. **Visualize**: Explore interactive graph representation
4. **Identify Issues**: View lint errors and code smells
5. **Prune Context**: Select subtree for focused refactoring
6. **Apply Refactoring**: Choose and execute refactoring patterns
7. **Review Changes**: Validate modifications before applying

Advanced Features

Custom Refactoring Rules

```python
class CustomRefactoring:
    def __init__(self, name: str, pattern: str, transform_fn):
        self.name = name
        self.pattern = pattern  # Cypher query pattern
        self.transform = transform_fn

    def apply(self, graph: Graph, node_id: str):
        # Find matching patterns
        matches = graph.run(self.pattern, node_id=node_id)

        # Apply transformation
        for match in matches:
            self.transform(match)
```

Integration with IDEs

- VS Code extension for in-editor refactoring
- JetBrains plugin support
- Language server protocol implementation

Performance Optimization

- Incremental parsing for large codebases

- Caching of AST results
- Parallel processing for multi-file projects

Conclusion

This tool provides a powerful approach to refactoring large codebases by:

- Using graph databases to manage complex relationships
- Providing visual navigation and context pruning
- Enabling algorithmic refactoring without LLM limitations
- Supporting multiple programming languages
- Running as a standalone desktop application

The combination of AST analysis, graph representation, and intelligent pruning creates an effective solution for managing and refactoring large-scale software projects.